Computer Lab 2: The Multiple Server Queue

Objectives

- Gain experience with Simkit
- Implement more complex model
- Re-use of `ArrivalProcess` class
- Communicating between objects with `SimEventListener`
- Communicate state changes by firing `PropertyChangeEvent`
- Use `SimpleStatsTimeVarying` to collect time-varying means

Description

In today's lab you will create a model of the Multiple Server (*G/G/k*) queue in Simkit. The model will re-use the Arrival process of Lab 01 and add the server functionality. You should not have to change any of the code in `ArrivalProcess`. You will write two classes: A `Server` class that will implement the server portion of the queue and a pure execution class (`GGk`) that will run the model for 1000 time units and collect statistics.

Writing the Server Class

The Event Graph to create the Server portion of the model is shown in Figure 1. The connection between the Arrival event of the `Arrival` component and the Arrival event of the `Server` component will be implemented using the `addSimEventListener` method, which will be described below.

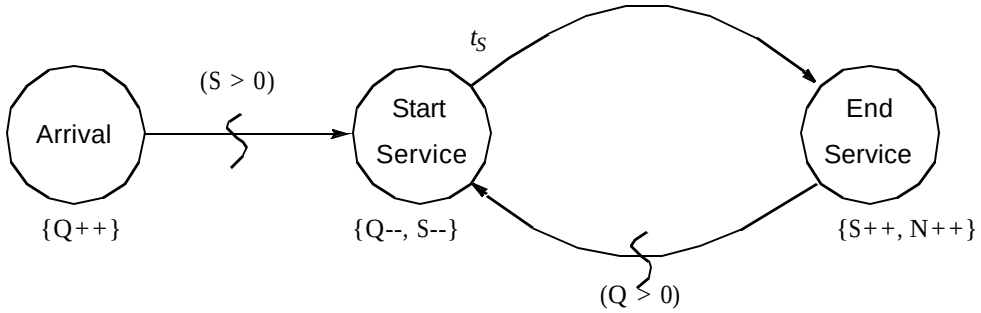

Figure 1. Server Event Graph[1]

Define a class in the `oa3302` package called `Server` extending `SimEntityBase`[2]. Define the instance variables for your parameters and state variables as shown in Table1 on page2. As with the `ArrivalProcess`, your state variables should have getters but no setters, and the parameters will have both setters and getters.

Next, write the first version of the "do" methods. As with the `ArrivalProcess`, each event in Figure 1 will correspond to a method with "do" prefixed ("`doArrival()`", "`doStartService()`", and "`doEndService()`"). You should also write a `doRun()` method, which will be discussed below.

Your `doArrival()` method should look like this (yours should be commented, of course):

1. The state variable N, incremented in the EndService event, is the number of customers who have been through the system.
2. Don't forget to `import simkit.*;`

Table 1: Parameters and State Variables for Server Class

Parameters	State Variables
`serviceTime (RandomVariate)`	`numberInQueue (int)`
`totalNumberServers (int)`	`numberAvailableServers (int)`
	`numberServed (int)`[a]

a. This corresponds to the state N in Figure 1.

```
public void doArrival() {
      firePropertyChange("numberInQueue", numberInQueue, ++numberInQueue);
      if (numberAvailableServers > 0) {
            waitDelay("StartService", 0.0);
      }
}
```

Note that the edge condition is implemented by wrapping the corresponding `waitDelay()` statement for the edge in an `if` test. Write the other "do" methods in a similar manner, making sure to associate each scheduling edge in Figure 1 with a call to `waitDelay()`. For now, hard-wire service times of 1.1; we'll add the randomness in a bit. Do not proceed until your class compiles.

The `reset()` method is necessary in this class for setting the initial values of `numberInQueue` (to 0) and `numberAvailableServers` (to `totalNumberServers`). Whenever the value of a state variable changes, a PropertyChange event should be fired, as in the `doArrival()` method above. However, the initial values of state variables are set in `reset()` and `firePropertyChange()` is called in `doRun()` for time-varying state variables (only). Thus, a part of your `reset()` method should look like this:

```
public void reset() {
      super.reset();
      numberInQueue = 0;
      numberAvailableServers = totalNumberServers;
...
}
```

Add similar code for `numberServed` in `reset()`. The `doRun()` method should only fire property changes for the time-varying state variables (*not* `numberServed`):

```
public void doRun() {
      firePropertyChange("numberInQueue", numberInQueue);
      firePropertyChange("numberAvailableServers", numberAvailableServers);
}
```

Note that this form of `firePropertyChange()` has signature `(String, int)`.

To complete the first iteration of your `Server` class, write a constructor that takes the total number of servers as an argument to its constructor and sets the `totalNumberServers` parameter to that value.

Now, write a pure execution class[1] called `GGk`. The `main` method should:

1. Instantiate an `ArrivalProcess` (call it "`arrival`"). You can hard-wire your parameters (make it Exponential with mean of 1.7 and seed of 12345).

1. That is, a class consisting only of a `main` method.

2. Instantiate a `Server` (call it "`server`"). Pass it "2" in the constructor (for the number of servers).
3. Add server as a SimEventListener to arrival as follows (in `main`):
```
arrival.addSimEventListener(server);
```
4. Invoke:
```
Schedule.stopAtTime(2.0);
Schedule.reset();
Schedule.setSingleStep(true);
Schedule.startSimulation();
```
5. Compile and run. Press Enter after each event. Your output should look like this:[1]

```
** Event List -- Starting Simulation **
0.000   Run
0.000   Run
2.000   Stop
 ** End  of Event List -- Starting Simulation **

Time: 0.000     Current Event: Run      [1]
 ** Event List --  **
0.000   Run
0.655   Arrival
2.000   Stop
 ** End  of Event List --  **

Time: 0.000     Current Event: Run      [2]
 ** Event List --  **
0.655   Arrival
2.000   Stop
 ** End  of Event List --  **

Time: 0.655     Current Event: Arrival  [1]
 ** Event List --  **
0.655   StartService
1.106   Arrival
2.000   Stop
 ** End  of Event List --  **

Time: 0.655     Current Event: StartService     [1]
 ** Event List --  **
1.106   Arrival
1.755   EndService
2.000   Stop
 ** End  of Event List --  **

Time: 1.106     Current Event: Arrival  [2]
 ** Event List --  **
1.106   StartService
1.755   EndService
2.000   Stop
3.866   Arrival
```

1. Note that there are two `Run` events at the beginning - one for `arrival` and one for `service`.

```
 ** End  of Event List --  **

Time: 1.106     Current Event: StartService     [2]
 ** Event List --  **
1.755   EndService
2.000   Stop
2.206   EndService
3.866   Arrival
 ** End  of Event List --  **

Time: 1.755     Current Event: EndService       [1]
 ** Event List --  **
2.000   Stop
2.206   EndService
3.866   Arrival
 ** End  of Event List --  **

Time: 2.000     Current Event: Stop     [1]
 ** Event List --  **
              << empty >>
 ** End  of Event List --  **
```

The `server` object has had its Arrival event triggered by the `arrival` object's Arrival event!

Now that you have a running program, you need to make the service times random. Add an instance variable of type `RandomVariate`[1] to generate the service times and add a `RandomVariate` argument to your constructor.

Modify the `waitDelay()` that schedules the EndService event to generate a random service time using the `serviceTime` object (just as in the `ArrivalProcess` class). You will have to change the instantiation in `main` to match the constructor.

You may hard-wire the service times in `main()` to use the gamma distribution. Recall that the gamma distribution has two parameters, α and β, and that the mean and variance for a gamma random variable are $\mu = \alpha\beta$ and $\sigma^2 = \alpha\beta^2$, respectively. The Gamma random variable generator in Simkit takes α and β as its parameters. So, in `main()` you will have to define an `Object[]` array containing them to pass to the `Server` constructor. One way to do this is:

```
Object[] parameters = new Object[] {new Double(2.5), new Double(1.2)};
```

Pass this as the second argument to `RandomVariateFactory`, with the string `"Gamma"` as the first argument. Use a seed of 54321, and two servers as above. Compile and execute to get the following output:

```
** Event List -- Starting Simulation **
0.000   Run
0.000   Run
2.000   Stop
 ** End  of Event List -- Starting Simulation **

Time: 0.000     Current Event: Run      [1]
 ** Event List --  **
0.000   Run
0.655   Arrival
2.000   Stop
```

1. Call it `serviceTime`

```
 ** End  of Event List --  **

Time: 0.000     Current Event: Run      [2]
 ** Event List --  **
0.655   Arrival
2.000   Stop
 ** End  of Event List --  **

Time: 0.655     Current Event: Arrival  [1]
 ** Event List --  **
0.655   StartService
1.106   Arrival
2.000   Stop
 ** End  of Event List --  **

Time: 0.655     Current Event: StartService     [1]
 ** Event List --  **
1.106   Arrival
1.505   EndService
2.000   Stop
 ** End  of Event List --  **

Time: 1.106     Current Event: Arrival  [2]
 ** Event List --  **
1.106   StartService
1.505   EndService
2.000   Stop
3.866   Arrival
 ** End  of Event List --  **

Time: 1.106     Current Event: StartService     [2]
 ** Event List --  **
1.505   EndService
2.000   Stop
3.866   Arrival
5.082   EndService
 ** End  of Event List --  **

Time: 1.505     Current Event: EndService       [1]
 ** Event List --  **
2.000   Stop
3.866   Arrival
5.082   EndService
 ** End  of Event List --  **

Time: 2.000     Current Event: Stop     [1]
 ** Event List --  **
              << empty >>
 ** End  of Event List --  **
```

Collecting Statistics

At this point your model is complete, and you can now write a program that runs it for awhile and collects some statistics. Simkit provides a class in the `simkit.stat` package called `SimpleStatsTimeVarying` that can estimate a time-varying mean from date. The mechanism for obtaining those

values from your model uses the `PropertyChangeEvent` that you fired at all your state changes. The three arguments of `firePropertyChange()` are as follows[1]:

- The name of the property.
- The old value of the property.
- The new value of the property.

Modify `GGk` to run your model and collect statistics. After instantiating the ArrivalProcess and Server objects, instantiate an object of type `SimpleStatsTimeVarying` as follows:

```
SimpleStatsTimeVarying niqStat = new SimpleStatsTimeVarying("numberInQueue");
```

The `String` passed to the constructor, `"numberInQueue"`, has the same name (case-sensitive) as the property that was fired in `Server`. Instantiate another one for `numberAvailableServers`. These should be done after the `Server` and `ArrivalProcess` objects are instantiated but before the `Simulation` methods are invoked. Finally, add each instance to the `server` as a PropertyChangeListener. For example,

```
server.addPropertyChangeListener(niqStat);
```

with the other being similar.

When this compiles and works for short runs, set it to stop at time 1000.0, set the verbose/single-step mode to `false`, and output the mean values after the run. Use the `getMean()` method of `SimpleStats` (see the javadoc for further details about `SimpleStatsTimeVarying`). The average utilization is defined to be 1.0 - (avg # available servers) / total number of servers. The code to echo back the parameters of the model should be written in `main` before `Schedule.startSimulation()`; the code to write the output statistics should also be in `main` but come after `Schedule.startSimulation()`. Your final output should look like this:[2]

```
Multiple Server Queue with 2 servers

Service time distribution is Gamma (2.5, 1.2)

Arrival Process with Exponential (1.7) interarrival times

Simulation ended at time         1000.0000

There have been 614 customers arrive to the system

There have been 607 customers served

Average Number in Queue          4.0739

Average Utilization              0.9166
```

To get this output, use the getter methods from `Server` as well as `getMean()` from `SimpleStatsTimeVarying`. The simulated time should be obtained using `Schedule.getSimTime()`.

To get the first two lines, write a method called `paramString()` in Server that returns a String as follows:

```
public String paramString() {
       return "Multiple Server Queue with " + totalNumberServers + " servers" +
```

1. There is additionally the two-parameter version you wrote in `doRun()`. The two-parameters version has just the name of the property and the new value.
2. Use `java.text.DecimalFormat` to format the numbers to the desired decimal places.

```
                "\n\nService time distribution is " + serviceTime;
    }
```

Write a similar `paramString()` method for `ArrivalProcess` to get the next line in the output.

Deliverables

Turn in the code for your `Server` and `GGk` classes and hard copy of your two outputs (the last verbose output and the one that collects statistics. You do not have to turn in the source code for your `ArrivalProcess` class from before.

Frequently Asked Questions

What does addSimEventListener do?

After the listenee executes an event from the Event List, it passes that event to the listener. If the listener has an event that matches, then that event is executed. In this program, the `Server` instance has its Arrival event triggered by the `ArrivalProcess`'s Arrival event.

What's with all this firePropertyChange stuff?

Simkit can exploit the JavaBeans property listener pattern by having only those objects who are "interested" in a given property registering that interest and receiving a `PropertyChangeEvent` when the property changes value. The `firePropertyChange()` method dispatches a `PropertyChangeEvent` to all registered listeners for the object with the property. Although this is a little more work now, the property change listener pattern makes things much easier down the road.